Complexidade de Algoritmos

Allan Robson

DCA

Estruturas de Dados

Complexidade de Algoritmos

- Análise de Algoritmos é uma área da Ciência da Computação que estuda os algoritmos.

- Busca responder a seguinte pergunta:

“Podemos fazer um algoritmo mais eficiente?”

- Podemos resolver um problema de várias maneiras diferentes, isto é, podemos utilizar algoritmos diferentes para um mesmo problema;

- Algoritmos diferentes para resolver o mesmo problema não necessariamente fazem com a mesma eficiência;

- Para comparar a eficiência dos algoritmos foi criada uma medida chamada de **Complexidade Computacional**

- A Complexidade Computacional indica o “custo” ao se aplicar um algoritmo;

$$\boldsymbol{custo = mem\acute{o}ria + tempo}$$

  - **Memória**: quando de espaço o algoritmo vai consumir;
  - **Tempo**: duração de execução

- Para medir a complexidade computacional de um algoritmo devemos **medir quantas instruções** são realizadas para a execução da tarefa.

Contando Instruções

- Tomando como exemplo um algoritmo que procure o maior elemento de um vetor:

```
int Maior = A[0];
for (i = 0; i < n; i++) {
    if (A[i] >= Maior) {
        Maior = A[i];
    }
}
```

- Para contar instruções devemos contar quantas "**instruções simples**" são executadas:

- Instrução simples:
  - Atribuição de valor a uma variável;
  - Comparação entre dois valores
  - Operações aritméticas básicas (soma, subtração, multiplicação, divisão)

Contando Instruções

- **Linha 1 (custo total 1)**:
  - 1 instrução;

```
1. int M = A[0];
2. for (i = 0; i<n; i++) {
3.     if (A[i] >= M) {
4.             M = A[i];
5.     }
6. }
```

Contando Instruções

- **Linha 2 (custo total $2n + 2$):**
  - Na primeira execução (custo 2):
    - Inicialização da variável "i"
    - Comparação "*i<n*"
  - Durante a execução do laço (custo $2n$):
    - Incremento "i++" (executado n vezes)
    - comparação "$i < n$" (executado n vezes)

```
1. int M = A[0];
2. for (i = 0; i<n; i++) {
3.     if (A[i] >= M) {
4.             M = A[i];
5.     }
6. }
```

Contando Instruções

- **Linhas 3 (custo total n)**:
  - Comparação "*A[i] >= M*"

```
1. int M = A[0];
2. for (i = 0; i<n; i++) {
3.     if (A[i] >= M) {
4.             M = A[i];
5.     }
6. }
```

Contando Instruções

- **Linhas 4 (custo total n)**:
  - Atribuição "*M = A[i]*"
  - No pior caso será realizada n atribuições

```
1. int M = A[0];
2. for (i = 0; i<n; i++) {
3.     if (A[i] >= M) {
4.         M = A[i];
5.     }
6. }
```

- O custo total do algoritmo é dado por:

$$f(n) = 1 + (2n + 2) + n + n$$

$$f(n) = 4n + 3$$

- Essa função dá uma ideia do custo de execução do algoritmo para um problema de tamanho n.

- **Será se todos os termos da função f são necessários para termos uma noção do custo?**

- De fato, nem todos os termos são necessários;

- Podemos descartar certos termos na função e manter apenas os que nos dizem o que acontece com a função quando o tamanho dos dados de entrada n cresce muito;

Comportamento Assintótico

- A ideia do **comportamento assintótico** é analisar como o algoritmo se comporta quando $\boldsymbol{n} \to \infty$

- Podemos descartar todos os termos que crescem lentamente e manter apenas os que crescem mais rápido;

- Para representar o comportamento assintótico de um algoritmo utilizaremos a notação ***big*-O.**

- A notação ***big*-O** representa o custo (seja de tempo ou de espaço) do algoritmo no **pior caso** possível para todas as entradas de tamanho n;

- Desse modo, podemos dizer que o comportamento do nosso algoritmo não pode nunca ultrapassar um certo limite.

- A função $f(n) = 4n + 3$ possui dois termos
  - O termo 3
  - O termo $4n$

- O termo 3 é uma constante, então não se altera à medida que n aumenta;

- Assim, nossa função pode ser reduzida para:
  - $\boldsymbol{f(n) = 4n}$

- A nossa nova função de custo $\boldsymbol{f(n) = 4n}$ possui apenas um único termo;

- O termo $4n$ pode ainda ser simplificado quando consideramos o comportamento assintótico da função;

- **Constantes que multiplicam o termo n também podem ser descartadas;**

- Ignorar essas constantes de multiplicação equivale a ignorar as particularidades de cada linguagem/compilador e analisar apenas a ideia do algoritmo;

- **Assim, utilizando a notação *big*-O, nossa função de custo pode ser reduzida para:**

$$\boldsymbol{f(n) = O(n)}$$

Comportamento Assintótico

- Na análise do comportamento assintótico apenas os termos de maior crescimento (maior expoente) são considerados.
- Todos os termos constantes e de menor crescimento são descartados;
- Exemplo:

Função Custo	Comportamento Assintótico
$f(n) = 105$	$f(n) = O(1)$
$f(n) = 15n + 2$	$f(n) = O(n)$
$f(n) = n^2 + 5n + 2$	$f(n) = O(n^2)$
$f(n) = 5n^3 + 200n^2 + 112$	$f(n) = O(n^3)$

- De modo geral, podemos obter a função de custo de um programa simples apenas **contando os comandos de laços aninhados**;
- **Algoritmos sem laço**: número constante de instruções (exceto se houver recursão)
  - $f(n) = O(1)$
- **Com um laço** indo de 1 a n:
  - $f(n) = O(n)$
- **Dois comandos de laço aninhados**:
  - $f(n) = O(n^2)$

Comportamento Assintótico: Outras classes de problemas

- $\boldsymbol{O(log(n))}$
  - Típica de algoritmos que resolvem um problema transformando-o em problemas menores;
  - Mais rápido que algoritmos $O(n)$.

- $\boldsymbol{O(n\,log(n))}$
  - Esses algoritmos resolvem um problema transformando-o em problemas menores, que são resolvidos de forma independente e depois unidos.

Comportamento Assintótico: Outras classes de problemas

- $\boldsymbol{O(2^n)}$ - Ordem exponencial
  - Geralmente ocorre quando se usa uma solução de "força bruta";
  - Não são úteis do ponto de vista prático.

- $\boldsymbol{O(n!)}$ - Ordem Fatorial
  - Geralmente ocorre quando se usa uma solução de "força bruta";
  - Não são úteis do ponto de vista prático;
  - Possui um comportamento muito pior que o exponencial.

Comportamento Assintótico: Outras classes de problemas

- Comparação no tempo de execução: Considerando um computador que seja capaz de executar um milhão de operações por segundo.

$f(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 50$	$n = 100$
n	$1{,}0E-05$ segundos	$2{,}0E-05$ segundos	$4{,}0E-05$ segundos	$5{,}0E-05$ segundos	$6{,}0E-05$ segundos
$nlog(n)$	$3{,}3E-05$ segundos	$8{,}6E-05$ segundos	$2{,}1E-04$ segundos	$2{,}8E-04$ segundos	$3{,}5E-04$ segundos
n^2	$1{,}0E-04$ segundos	$4{,}0E-04$ segundos	$1{,}6E-03$ segundos	$2{,}5E-03$ segundos	$3{,}6E-03$ segundos
n^3	$1{,}0E-03$ segundos	$8{,}0E-03$ segundos	$6{,}4E-02$ segundos	$0{,}13$ segundos	$0{,}22$ segundos
2^n	$1{,}0E-02$ segundos	$1{,}0$ segundos	$2{,}8$ dias	$35{,}7$ anos	$365{,}6$ séculos